Trabajo Práctico 1
Programación Funcional

Paradigmas ~~de Lenguajes~~ de Programación
2do cuatrimestre 2024

Fecha de entrega: 20 de septiembre

Introducción

Queremos definir un tipo de funciones, llamadas 'procesadores', que nos van a permitir recorrer y procesar distintas estructuras, y obtener resultados a partir de ellas.

Un procesador se define con el siguiente tipo: `type Procesador a b = a ->[b]`, donde `a` es el tipo de la estructura y `b` es el tipo de los resultados a generar (pueden devolverse varios, uno, o ninguno, de acuerdo al procesador definido).

Estructuras a utilizar

Para este trabajo se utilizarán las siguientes definiciones de tipos de datos:

- Árboles ternarios (`AT`) y *rose trees* (`RoseTree`) siguen la definición ya conocida:

```
data AT a = Nil | Tern a (AT a) (AT a) (AT a)
data RoseTree a = Rose a [RoseTree a]
```

- Tries (del inglés *reTRIEval*), que ya los conocen de AED2/AED, siguiendo una implementación *naive*, que puede ser utilizada tanto como conjunto o como diccionario:

```
data Trie a = TrieNodo (Maybe a) [(Char, Trie a)]
```

  Como conjunto, `a` podría ser cualquier tipo, donde cualquiera de sus valores indicaría que el camino recorrido hasta el momento es una palabra del conjunto (e.g., podría ser un `[Char]` y colocar la palabra entera registrada, o usarse `Bool` y usar sólo el valor `True`, por claridad), mientras que como diccionario el tipo `a` sería el del *valor*, siendo la *clave* el camino recorrido hasta llegar a ese nodo. En ambos casos, si el valor del nodo es `Nothing`, esto denota que el camino recorrido hasta ese nodo no pertenece al conjunto o no es una clave definida en el diccionario, respectivamente.

  Decimos que es una implementación *naive*, en tanto no se puede asegurar que un carácter esté definido sólo una vez en la lista. Esto no genera problemas al momento de definir las palabras, ya que alcanza con que estén definidas por algún camino, aunque sí se pierden algunas propiedades interesantes de los tries (e.g., que tenga costo temporal constante el ubicar una clave, dado un alfabeto finito).

Ejercicios

Procesamiento básico de estructuras

Primero arrancaremos creando algunas funciones para generar salidas para algunas estructuras conocidas.

Ejercicio 1

Se pide implementar las siguientes funciones procesadoras:

a) `procVacio :: Procesador a b`, que para cualquier estructura devuelve un resultado vacío.

b) `procId :: Procesador a a`, que devuelve un único resultado, que contiene la estructura original completa.

c) `procCola :: Procesador [a] a`, que devuelve todos los elementos de la lista sin incluir al primero. Si la lista estuviera vacía, no se deben devolver resultados (i.e., el resultado final del procesador es una lista vacía).

d) `procHijosRose :: Procesador (RoseTree a) (RoseTree a)`, que dado un RoseTree devuelve como resultados todos sus hijos (i.e., los RoseTrees que salen directamente de la raíz).

e) `procHijosAT :: Procesador (AT a) (AT a)`, análogo a la anterior, pero para árboles ternarios (i.e., se deben devolver tres resultados si el árbol tiene raíz, y ninguno si es `Nil`).

f) `procRaizTrie :: Procesador (Trie a) (Maybe a)`, que devuelve el valor del nodo raíz del Trie.

g) `procSubTries :: Procesador (Trie a) (Char, Trie a)`, que devuelve todos los pares de carácter y Trie, que corresponden con las aristas y el nodo hijo del Trie al cual se dirigen.

Uso de esquemas de recursión

Ahora vamos a recorrer las distintas estructuras usando sus esquemas de recursión estructural. Como vimos anteriormente, para las listas ya existe `foldr`, aunque vamos a necesitar los esquemas para el resto de las estructuras.

Ejercicio 2

Implementar las funciones `foldAT`, `foldRose` y `foldTrie` para recorrer árboles ternarios, RoseTrees y Tries, respectivamente. Indicar explícitamente el tipo de cada función. En estos casos, sí se puede usar recursión explícita.

Procesamiento de listas

Ejercicio 3

Definir los siguiente procesadores para listas:

a) `unoxuno :: Procesador [a] [a]`, que devuelve cada elemento de la lista en una lista singleton.

E.g., `unoxuno [3,1,4,1,5,9]` $\rightsquigarrow$ `[[3],[1],[4],[1],[5],[9]]`

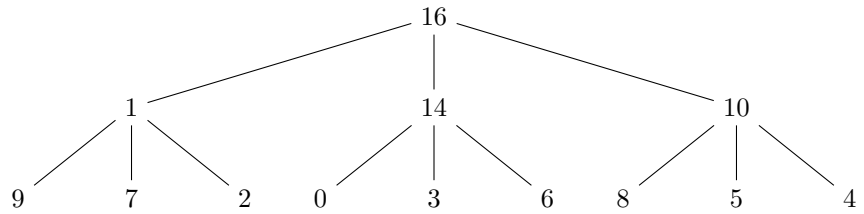
b) `sufijos :: Procesador [a] [a]`, que devuelve todos los sufijos de la lista que se está procesando.

E.g., usando la notación de strings para `[Char]`, `sufijos "Plp"` $\rightsquigarrow$ `["Plp", "lp", "p", ""]`

Procesamiento de árboles

Ejercicio 4

Implementar los procesadores para hacer los recorridos conocidos sobre árboles ternarios: `preorder`, `postorder` e `inorder`, indicando su tipo. E.g., dado el siguiente árbol ternario `at`:



los resultados de sus respectivos recorridos son:

- `preorder at` ⇝ `[16,1,9,7,2,14,0,3,6,10,8,5,4]`
- `postorder at` ⇝ `[9,7,2,1,0,3,6,14,8,5,4,10,16]`
- `inorder at` ⇝ `[9,7,1,2,0,3,14,6,16,8,5,10,4]`

Ejercicio 5

Definir los siguientes procesadores de RoseTrees:

a) `preorderRose :: Procesador (RoseTree a) a`, que devuelve el recorrido en profundidad del `RoseTree` (análogo al `preorder` anterior, pero para RoseTrees).

b) `hojasRose :: Procesador (RoseTree a) a`, que devuelve las hojas del RoseTree procesado (i.e., los nodos que no tienen hijos).

c) `ramasRose :: Procesador (RoseTree a) [a]`, que devuelve los caminos desde la raíz hasta las hojas del RoseTree.

Procesamiento de Tries

Ejercicio 6

Definir la función `caminos` que, dado un Trie, devuelva las cadenas que denotan un camino entre la raíz y cada uno de los nodos. No se deben incluir caminos repetidos.

E.g., dado el siguiente Trie:

```
t = TrieNodo Nothing
     [ ('a', TrieNodo (Just True) []),
       ('b', TrieNodo Nothing
         [('a', TrieNodo (Just True)
           [('d', TrieNodo Nothing [])])
         ]),
       ('c', TrieNodo (Just True) [])
     ]


caminos t ⇝ ["", "a", "b", "ba", "bad", "c"]
```

Ejercicio 7

Definir la función `palabras` que, dado un Trie, devuelva las palabras que incluye. Una palabra estará incluida si y solo si en el nodo que termina un camino desde la raíz se encuentra un valor que no sea `Nothing`. No se deben incluir palabras repetidas.

E.g., dado el Trie anterior:

`palabras t` ⇝ `["a", "ba", "c"]`

Ejercicio 8

Implementar las siguientes operaciones sobre procesadores.

a) `ifProc :: (a->Bool) -> Procesador a b -> Procesador a b -> Procesador a b`,

Que aplica dos procesadores de acuerdo a una condición definida. Si la estructura cumple la condición, entonces se aplica el primer procesador y, en caso contrario, le aplica el segundo.

E.g., `ifProc esNil procVacio procId`, que es un procesador de árboles binarios que en caso de que el árbol fuera `Nil` se devolverá `[]`, y, si no, se devolverá una lista de un solo elemento con el árbol completo.

b) `(++!) :: Procesador a b -> Procesador a b -> Procesador a b`,

Que aplica dos procesadores a una misma estructura, y concatena los resultados en orden.

E.g., dado `at` el árbol ternario del ejemplo del ejercicio 4 :

`(postorder ++! preorder) at` $\rightsquigarrow$

`[9,7,2,1,0,3,6,14,8,5,4,10,16,16,1,9,7,2,14,0,3,6,10,8,5,4]`

c) `(.!) :: Procesador b c -> Procesador a b -> Procesador a c`,

Que compone dos procesadores de la siguiente manera:
Dada una estructura de tipo `a`, se le aplica el segundo procesador, obteniendo una lista de resultados de tipo `b`, y a cada uno de estos resultados se le aplica el primer procesador, y luego se unen los resultados finales en una lista de elementos de tipo `c`.

E.g., `((\z->[0..z]) .! (map (+1))) [1,3]` $\rightsquigarrow$ `[0,1,2,0,1,2,3,4]`

Ejercicio 9

De acuerdo a las definiciones de las funciones para árboles ternarios de más arriba, se pide demostrar lo siguiente:

$\forall$ `t :: AT a` . $\forall$ `x :: a` . (`elem x (preorder t)` = `elem x (postorder t)`)

Pautas de Entrega

Se debe entregar a través del campus un único archivo llamado "tp1.hs" conteniendo el código con la implementación de las funciones pedidas. Para eso, ya se encuentra disponible la entrega "TP1 - Programación Funcional" en la solapa "TPs" (configurada de forma grupal para que sólo una persona haga la entrega en nombre del grupo). El código entregado **debe** incluir tests que permitan probar las funciones definidas. El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: Es necesario para los ejercicios que usen las funciones que vimos en clase y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Tests: cada ejercicio debe contar con uno o más ejemplos que muestren que exhibe la funcionalidad solicitada. Para esto se recomienda la codificación de tests usando el paquete HUnit https://hackage.haskell.org/package/HUnit. El esqueleto provisto incluye algunos ejemplos de cómo utilizarlo para definir casos de test para cada ejercicio.

Para instalar HUnit usar: `> cabal install --lib HUnit`

Para instalar cabal ver: https://wiki.haskell.org/Cabal-Install

Para ejecutar los tests usar: `> main` para todos los tests, `> do runTestTT testsEjN` para los tests del ejercicio N.

Importante: Se espera que la elaboración de este trabajo sea 100 % de los estudiantes del grupo que realiza la entrega. Así que, más allá de que pueden tomar información de lo visto en las clases o consultar información en la documentación de Haskell o disponible en Internet, no se podrán utilizar herramientas para generar parcial o totalmente en forma automática la resolución del TP (e.g., chat-GPT, copilot, etc). En caso de detectarse esto, el trabajo será considerado como un plagio, por lo que será gestionado de la misma forma que se resuelven las copias en los parciales u otras instancias de evaluación.

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquéllos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.